Secure Compilation of Object-Oriented Components to Protected Module Architectures*

Marco Patrignani, Dave Clarke, and Frank Piessens

iMinds-DistriNet, Dept. Computer Science, KU Leuven
{first.last}@cs.kuleuven.be

Abstract. A fully abstract compilation scheme prevents the security features of the high-level language from being bypassed by an attacker operating at a particular lower level. This paper presents a fully abstract compilation scheme from a realistic object-oriented language with dynamic memory allocation, cross-package inheritance, exceptions and inner classes to untyped machine code. Full abstraction of the compilation scheme relies on enhancing the low-level machine model with a fine-grained, program counter-based memory access control mechanism. This paper contains the outline of a formal proof of full abstraction of the compilation scheme. Measurements of the overhead introduced by the compilation scheme indicate that it is negligible.

1 Introduction

Modern high-level languages such as ML, Java or Scala offer security features to programmers in the form of type systems, module systems, or encapsulation primitives. These mechanisms can be used as security building blocks to withstand the threat of attackers acting at the high level. For the software to be secure, attackers acting at lower levels need to be considered as well. Thus it is important that high-level security properties are preserved after the high-level code is compiled to machine code. Such a security-preserving compilation scheme is called *fully abstract* [1]. An implication of such a compilation scheme is that the power of a low-level attacker is reduced to that of a high-level one. The notion of fully abstract compilation is well suited for expressing the preservation of security policies through compilation, as it preserves and reflects contextual equivalence. Two programs are contextually equivalent if they cannot be distinguished by a third one. Contextual equivalence models security policies as follows: saying that variable f of program C is confidential is equivalent to saying that C is contextually equivalent to any program C' that differs from C in its value for f. A fully abstract compilation scheme does not eliminate high-level

* This work has been supported in part by the Intel Lab'sUniversity Research Office. This research is also partially funded by the Research Fund KU Leuven, and by the EU FP7 project NESSoS. With the financial support from the Prevention of and Fight against Crime Programme of the European Union (B-CCENTRE). Marco Patrignani holds a Ph.D. fellowship from the Research Foundation Flanders (FWO).

security flaws. It is, in a sense, conservative, introducing no more vulnerabilities at the low level than the ones already exploitable at the high level.

Fully abstract compilation of modern high-level languages is hard to achieve. Compilation of Java to JVM or of C$^{\#}$ to the .NET framework [12] are some of the examples where compilation is not fully abstract. Recent techniques that achieve fully abstract compilation rely on address space layout randomisation [2,9], type-based invariants [4,7], and enhancing the low-level machine model with a fine-grained program counter-based memory access control mechanism [3].

The threat model considered in this paper is that of an attacker with low-level code execution privileges. Such an attacker can inject and execute malicious code at machine level and violate the security properties of the machine code generated by the compiler. In order to withstand such a low-level attacker, high-level security features must be preserved in the code generated during compilation. Agten *et al.* [3] were the first to show that fully abstract compilation of a safe high-level programming language to untyped machine code is possible. They achieved this by enhancing the low-level machine model with a fine-grained program counter-based memory access control mechanism inspired by existing systems [14,15,17,21,22] and recent industrial prototypes [16]. One limitation of the work of Agten *et al.* is that it only considers a toy high-level language. The main contribution of this paper is showing how essential programming language features can be securely compiled to the same low-level machine model of Agten *et al.* The adopted low-level model is similar to a modern processor, so the compilation scheme handles subtleties such as flags and registers that an implementation would have to face. More precisely, this paper makes the following contributions:

- a secure compilation scheme from a model object-oriented language with dynamic memory allocation, cross-package inheritance, exceptions and inner classes to low-level, untyped machine code;
- the outline of a formal proof of full abstraction for this compilation scheme;
- measurements of the run-time overhead introduced by the compilation scheme.

The paper is organised as follows. Section 2 introduces background notions. Section 3 presents a secure compilation scheme for a language with dynamic memory allocation, cross-package inheritance, exceptions and inner classes. Section 4 outlines the proof of full abstraction of the compilation scheme. Section 5 presents benchmarks of the overhead introduced by the compilation scheme. Section 6 discusses limitations of the compilation scheme. Section 7 presents related work. Section 8 discusses future work and concludes.

2 Background

This section describes the low-level protection mechanism and the secure compilation scheme of Agten *et al.* [3], which is the starting point of this paper. Then the high-level language targeted by the compilation scheme is presented.

2.1 Low-level Model

To model a realistic compilation scheme, the targeted low-level language should be close to what is used by modern processors. For this reason this paper adopts a low-level language that models a standard Von Neumann machine consisting of a program counter, a registers file, a flags register and memory space [3].

In order to support full abstraction of the compilation scheme, the low-level language is enhanced with a protection mechanism: a fine-grained, program counter-based memory access control mechanism inspired by existing systems [14,15,17,21,22] and recent industrial prototypes [16]. We review this addition from the work of Agten *et al.* [3] and Strackx and Piessens [22]. This mechanism assumes that the memory is logically divided into a protected and an unprotected section. The protected section is further divided into a code and a data section. The code section contains a variable number of entry points: the only addresses to which instructions in unprotected memory can jump and execute. The data section is accessible only from the protected section. The size and location of each memory section are specified in a memory descriptor. The table below summarises the access control model enforced by the protection mechanism.

From\ To	Protected			Unprotected
	Entry Point	Code	Data	
Protected	r x	r x	r w	r w x
Unprotected	x			r w x

This protection mechanism provides a secure environment for code that needs to be protected from a potentially malicious surrounding environment. It is appealing in the context of embedded systems, where kernel-level protection mechanisms are often lacking.

2.2 A Secure Compiler for a Simple Language

Agten *et al.* [3] presented a secure (fully abstract) compilation scheme for a simple object based language. In an effort to be self-contained, this paper summarises their key points.

General notions. In the work of Agten *et al.*, programs consist of a single object with fields and methods declarations which are compiled to the protected memory partition. Compiled programs must be indistinguishable from the size point of view, thus a constant amount of space is reserved for each program, independent of its implementation. All methods and fields are sorted alphabetically. Thus equivalent compiled programs cannot be distinguished based on the ordering of low-level method calls. Methods and fields are given a unique index, starting from 0, based on their order of occurrence. Those indexes serve as the offset used to access methods and fields. Parameters and local variables are also given method-local indexes to be used as above.

Registers $\mathtt{r}_0$ to $\mathtt{r}_3$ are used as working registers for low-level instructions and registers $\mathtt{r}_4$ to $\mathtt{r}_{11}$ are used for parameters. The call stack is split into a protected

and an unprotected part, the former is allocated in the protected memory partition. A shadow stack pointer that points to the base of the protected stack is introduced to implement stack switches. When entering the protected memory, the protected stack is set as the active one; when leaving it, the unprotected stack is set to be the active one. To prevent tampering with the control flow, the base of the protected stack points to a procedure that writes 0 in $\texttt{r}_0$ and `halt`s. For each method, a prologue and an epilogue are appended to the method body. They allocate and deallocate activation records on the secure stack, The program counter is initialised to a given address in unprotected memory.

Entry points. For each method, an entry point in protected memory is created. Additionally, in order to enable returnbacks (returns from callbacks, which are calls to external code), a returnback entry point is created. Entry points act as proxies to the actual method implementations and are extended with security routines and checks.

These security routines reset unused registers and flags when leaving the protected memory to prevent them conveying unwanted information. For example, a callback to a function with two arguments resets all registers but $\texttt{r}_4$ and $\texttt{r}_5$ since they are the only ones that carry desired information. Checks are made to ensure that primitive-typed parameters have the right byte representation, e.g. `Unit`-typed parameters must have value 0, the chosen value of `Unit` type.

2.3 High-level Language

The high-level language targeted by this paper is Jeffrey and Rathke's Java Jr. [11]. Java Jr. is a strongly-typed, single-threaded, component-based, object-oriented language that enforces `private` fields and `public` methods. Java Jr. supports all the basic constructs one expects from a modern programming language, including dynamic memory allocation. A program in Java Jr., called a component, is a collection of sealed packages that communicate via interfaces and public objects. Java Jr. enforces a partition of packages into *import* and *export* ones. Import packages are analogous to the `.h` header file of a C program; they define *interfaces* and *externs*, which are references to externally defined objects. Export packages define *classes* and *objects*; they provide an implementation of an import package. Listing 1.1 illustrates the package system of Java Jr.

Listing 1.1 contains two package declarations: `PI` is an import package and `PE` is an export package implementing `PI`. Object `extAccount` allocated in `PE` provides an implementation for the `extern` with the same name defined in `PI`.

In Java Jr., primitive values, types and operations on them are assumed to be provided by a `System` package, whose name is omitted for the sake of brevity. The only primitive type is `Unit`, inhabited by `unit`. Since the focus of this paper is security, we write access modifiers for methods and fields even though the syntax of Java Jr. does not require them.

The security mechanism of Java Jr. is given by `private` fields. In Java Jr., classes are private to the package that contains their declarations. Objects are allocated in the same package as the class they instantiate. Due to this package

```
1  package PI;
2    interface Account {
3      public createAccount() : Foo;
4      public getBalance() : Int;
5    }
6    extern extAccount : Account;
7
8  package PE;
9    class AccountClass implements PI.Account {
10     AccountClass() { counter = 0; }
11     public createAccount() : Account { return new PE.AccountClass(); }
12     public getBalance() : Int { return counter; }
13     private counter : Int;
14   }
15   object extAccount : AccountClass;
```

Listing 1.1. Example of the package system of Java Jr.

system, for a package to be compiled it only needs the import packages of any package it depends on. As a result, formal parameters in methods have interface types, since classes that implement those interfaces are unknown. Additionally, since constructors are not exposed in interfaces, cross-package object allocation must be through factory methods. For example, the name of class `AccountClass` from Listing 1.1 is not visible from outside package `PE`, thus expressions of the form `new PE.AccountClass()` cannot be written outside `PE`.

Java Jr. was chosen since it provides a clear notion of encapsulation for a high-level component, which makes for simpler reasoning about the secure compilation scheme. This allows us to pinpoint what the key insights are to achieve secure (fully abstract) compilation, so that they can be used when the language is extended with cross-package inheritance, exceptions and inner classes.

3 Secure Compilation of Java Jr.

After a series of examples describing possible attacks on a naïve compilation scheme, this section describes what is needed in order to provide a secure compiler for Java Jr., starting from the secure compiler described in Section 2.2, and extend it to support cross-package inheritance, exceptions and inner classes.

The following examples use some standard assumptions about how objects are compiled [6]. When an object is allocated, a word is reserved to indicate its class, which is used to dynamically dispatch methods. Fields are accessed via offsets and methods are dispatched based on offsets.

Example 1 (Type of the current object) *Suppose the compiled program includes two classes:* `Pair` *and* `Caesar`*. Class* `Pair` *implements pairs of* `Int`*eger values with two fields* `first` *and* `second`*, with getters and setters for them, method* `getFirst()` *returns the value of field* `first`*. Class* `Caesar` *implements a caesar cypher. It has a single* `Int`*eger field* `key` *and a method* `encrypt(v:Int)` *that returns value* `v` *encrypted with* `key`*. The* `key` *of the* `Caesar` *cypher is not accessible outside the class (i.e. it is private).*

The `key` *cannot be leaked at the high level, since high-level programs are strongly typed, but it can be leaked to low-level programs. A low-level, external program can perform a call to method* `getFirst()` *on an object of type* `Caesar`*; this will return the* `key` *field, since fields are accessed by offset. As low-level code is untyped, nothing prevents this attack from happening.*

Example 2 (Type of the arguments) *Similarly to Example 1, arguments of methods can be exploited in order to mount a low-level attack. Extend the program of Example 1 with another class* `ProxyPair` *with a method* `takeFirst(v:Pair)` *that returns* `getFirst()` *on the* `Pair` *object* `v`*. At the high level, this code gives rise to no attacks. At the low level, this code can be used to mount the following attack: if an object of type* `Caesar` *is passed as argument to method* `takeFirst( )`*, the code will leak the* `key`*.*

Example 3 (Leakage of object references) *Object references at the low-level are the address where objects are allocated. The attacker can call methods on objects it does not know of by guessing the address where an object is allocated. Passing object addresses from a secure program to an external one can also give away the allocation strategy of the compiler, as well as the size of allocated objects. An attacker that learns this information can then use it to mount attacks such as those presented in Example 1 and 2. From a technical point of view this means that leaking object addresses and accepting guessed addresses breaks full abstraction of the compilation scheme.*

3.1 A Secure Compiler for Java Jr.

Before proposing countermeasures to the attacks just listed, this section lists the modifications to the scheme of Section 2.2 that are needed in order to support compilation of Java Jr. and, more generally, of object-oriented programs.

Compilation of OO Languages. Fig. 1 shows a graphical representation of the protected memory section which is generated when securely compiling a Java Jr. component. Only a single protected memory section is needed, and all classes, objects and methods defined in the component are placed there. The protected code section contains entry points, described below, method body

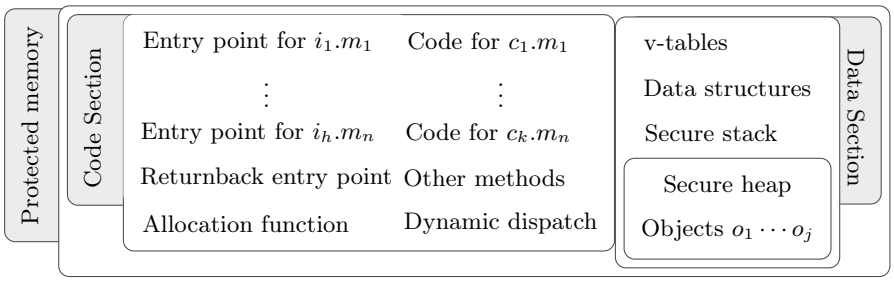

Fig. 1. Graphical representation of a compiled program.

implementations, the procedure for object allocation and the dynamic dispatch. The protected data section contains the v-tables, support data structures, a secure stack and a secure heap. The v-tables are data structures used to perform the dynamic dispatch of method calls; they associate the address of the method to be executed on an object based on its type and the method name. Data structures are defined in the remainder of this section and in Section 3.3.

In order to specify how the component interacts with external code, assume the component being compiled provides one import package without a corresponding export one. Refer to this package as *the distinguished import package (DIP)*. The DIP contains interface and extern definitions, thus callbacks are calls on methods defined in the DIP. Component code is assumed not to implement interfaces defined in the DIP, while external code which provides an implementation for the DIP can also implement interfaces defined in the component. Assuming the calling convention with the outer world is known, dynamic dispatch can easily take care of external objects whose classes implement interfaces defined in the component. Method call implementation adopted by external code is more complex since function calls must jump to the correct entry point, but it still can be achieved for example using object wrappers.

Finally, register $\mathtt{r}_4$ is used to identify the current object (`this`) in a method call at the low level. Before a callback, $\mathtt{r}_4$ is stored in the secure stack so as to be able to restore `this` to the right value once the callback returns.

Securing the Compilation. Following are the countermeasures added to withstand the attacks described in the Examples above. Since the countermeasure to Example 3 affects the others, it is presented first.

Object identity. To mask low-level object identities, a data structure $\mathcal{O}$ is added to the data structures of Fig. 1. It is a map between low-level object identities that have been passed to external code and natural numbers. Object identities that are passed to external code are added to $\mathcal{O}$ right before they are passed. The index in the data structure is then passed in place of the object identity, the same index must be passed for an already recorded object. Indices in $\mathcal{O}$ are thus passed in a deterministic order, based on the interaction between external and internal code. Code at entry points is responsible for retrieving object identities from $\mathcal{O}$ before the actual method call. As the only objects in the data structures are the ones the attacker knows, it cannot guess object identities.

Entry points. To support programming to an interface, the compilation scheme creates *method entry points* in protected memory for all interface-declared methods. A single *returnback entry point* for returning after a callback is also needed. Table 1 describes the code executed at those points. Both entry points are logically divided in two parts. The first part performs the checks described in the previous paragraph and then jumps either to the code that performs the dynamic dispatch or to the callback. The second part returns control to the location from which the entry point was called; call this the *exit point* for method entry points and *re-entry point* for the returnback entry point. For method calls to be well-typed, the code at entry points performs dynamic typechecks. This checks that a method is invoked on objects of the right type (line 2), with parameters of the

Method p entry point		Returnback entry point	
1	Load current object $v = \mathcal{O}(\mathsf{R}_4)$	a	Push current object $v = \mathsf{R}_4$, return address a and return type m
2	Check that v's class defines method p		
3	Load parameters $\overline{v}$ from $\mathcal{O}$	b	Reset flags and unused registers
4	Dynamic typecheck	c	Replace object identities with index in $\mathcal{O}$
5	Perform dynamic dispatch	d	Jump to callback address
Exit point (*run method code*)		Re-entry point (*run external code*)	
6	Reset flags and unused registers	e	Pop return type m and check it
7	Replace object identities with index in $\mathcal{O}$	f	Dynamic typecheck
		g	Pop return address a, current object v and resume execution

Table 1. Pseudo code executed at entry points. Loading means that a value is retrieved from the memory, push and pop are operations on the secure stack.

right type (line 4). Similar checks are executed when returning from a callback, in the returnback entry point (line f). These checks are performed only on objects whose class is defined in the compiled component, as they are allocated in protected memory; no control over externally allocated objects can be assumed. If any check fails, all registers and flags are cleared and the execution `halts`. Resetting flags and registers and `Unit`-typed value checks are as in Section 2.2. Dynamic typecheck involves checking primitive-typed values. These are needed for all primitive types inhabited by a finite number of values, such as `Unit` and `Bool`. For example, `bool`-typed parameters must have either value 1 or 0, which correspond to the high-level values `true` or `false` [7].

Insights. Following are the insights gained from developing a secure compilation scheme for Java Jr.; they will be useful in the following sections.

- Internal objects that are passed to external code must be remembered; their address must be masked.
- Strong typing of methods must be enforced with additional runtime checks.
- The low-level code must not introduce additional functionality (low-level functions in entry points) that is not available at the high level.

3.2 Secure Compilation of Cross-package Inheritance

Cross-package inheritance arises whenever class `D` from an export package `PSUB` `extends` class `C` from a different export package `PSUP`, as in Listing 1.2. Cross-package inheritance is not provided by Java Jr., as it would break the main result proven in the Java Jr. paper [11]. In order to allow cross-package inheritance, classes that can be extended must appear in import packages. Thus, given an import package, entry points are created not only for interface-defined methods, but also for class-defined ones and for constructors. Class `D` can optionally override methods of the super class `C`, as is the case with method `m()`. Within those methods, calls to `super` can be used in order to call method `m()` of the super class `C`. Alternatively, if a method is not overridden, such as method `z()`, calling d`.z()` on an object d of type `D` executes method `z()` defined in the super class `C`.

```
1 package PSUP;
2   class C {            // called the super class
3     public m():Int { ··· }
4     public z(): Int { ··· }
5   }
6 package PSUB;
7   class D extends PSUP.C {    // called the sub class
8     public m():Int{ super.m(); ··· }
9   }
```

Listing 1.2. Example of cross-package inheritance.

If the normal compilation scheme were followed, at the low-level d is allocated to a single memory area where fields from subobjects `C` and `D` are both allocated. Example 4 highlights the problems that arise in this setting.

Example 4 (Allocation of d) *Consider the case when `C` is protected and `D` is not. If d is allocated outside the protected memory partition, private fields of the `C` subobject become accessible to external code. If d is allocated inside the protected memory partition, two options arise. The first one is placing untrusted methods of `D` in the protected memory partition, violating the security of the compilation scheme. Otherwise, if methods of `D` are placed in the unprotected memory partition, they cannot access `D`'s fields via offset. Getters and setters for fields of `D` could be exposed through entry points, but this would violate full abstraction, as those methods are not available at the high level.*

The problems just presented above also arise when `C` is not protected but `D` is, thus compilation of cross-package inheritance cannot be achieved normally.

To allocate d securely, it is split in two sub-objects: dc, with fields of class `C`, and dd, with fields of class `D`; the object identity of d is dd [23].

Consider firstly the case when `C` is protected and `D` is not. External code needs to compile the expression d `= new D()` so that it calls `new C()` to create object dc in the protected memory section. External code must then save the resulting identifier for dc to perform `super` calls, since they are translated as method calls. The additional checks inserted at entry points presented in Section 3.1 ensure that `super` calls are always well-typed.

Consider then the case when `C` is not protected and `D` is. The secure compiler needs to call `new D()` and save the returned object identity for dd in a memory location, since `super` calls in this case are compiled as callbacks. When expression d `= new D()` is compiled, the unprotected address dc is stored at the low-level, right after the type of dd. The expression `super.m()` is compiled as dc`.m()`.

The creation of two separate objects may seem to break full abstraction of the compilation scheme in a way similar to what Abadi found out for inner classes [1]. In fact, low level external code is given the functionality to call dc`.m()`, which is not explicitly possible in the high-level language. However, d`.super.m()` is an implicit call to the `m()` method of `C`, functionality that the high-level language already has. Handling of cross-package inheritance does not add functionality at the low level, so it does not break full abstraction of the compilation scheme.

3.3 Exceptions

Secure compilation of languages supporting exceptions must handle the difficulties that result from the modification of the flow of execution of a program.

```
1  package P1;
2    class G {
3      public m():Void{
4        try{ new P2.H().e(); } catch (e : P3.MyException){ // handle e   } }
5    }
6  package P2;
7    class H {
8      public e():Void throws P3.MyException { throw new P3.MyException(); }
9    }
10 package P3;
11   class MyException implements Throwable {···}
```

Listing 1.3. Example of exceptions usage.

Exception handling can be securely implemented by modifying the runtime of the language so that it knows where to dispatch a thrown exception. Activation records are responsible for pointing to the exception handlers in order to propagate a thrown exception to the right handler. In Listing 1.3, the `catch` block of method `m()` in class `G` defines a handler for exceptions of type `MyException`. When the activation record for `m()` is allocated, the handler is registered. When an exception of type `MyException` is thrown, the stack is traversed to find the closest handler for exceptions of type `MyException`. As activation records are traversed and a handler is not found, those records are popped from the stack.

In the context of secure compilation, exception handlers are compiled in the usual manner. In order to implement throwing an exception in secure code that is caught in insecure code (or vice versa), throwing is compiled as callbacks (or calls). Thus two additional entry points are created: the *throw entry point* and the *throwback entry point*. These entry points forward calls to the secure and insecure exception dispatchers, respectively. The secure exception dispatcher traverses the secure stack looking for handlers for the thrown exception. After an activation record has been inspected and deallocated, the exception is forwarded to the external code through the throwback entry point. In order to prevent exploits similar to those of Example 2, the throwback entry point must remember internally allocated exceptions that are thrown to external code. So, a data structure $\mathcal{E}$, similar to $\mathcal{O}$, is created to register leaked exceptions. This prevents external code from passing a fake object identity to the secure exception handler in place of the object identity of an exception, effectively throwing a non-existent exception. External code can implement a wrapper around the exception object identity in order to be able to associate it to its type and then be able to recognise the type of the exception in the handler.

Fig. 2 presents a graphical overview of how exceptions are handled normally (on the left) and in the presented compilation scheme (on the right). Lower case letters indicate the allocation record for the corresponding function. A subscript s indicates a secure function; the stack grows downward. The order in which exception handlers are searched is indicated on arrows. The throw and throwback entry point split the same call in two parts.

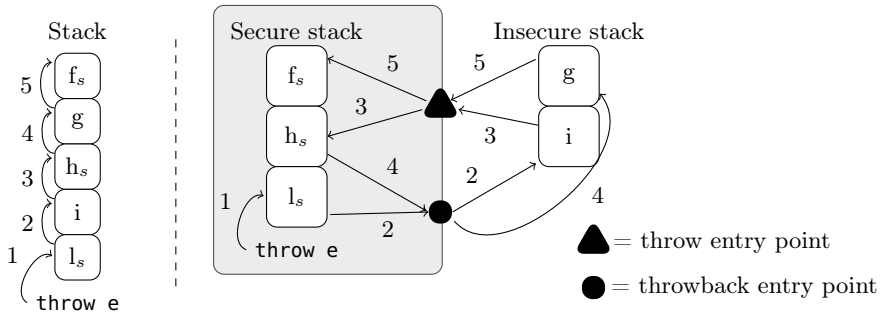

Fig. 2. Comparison of ways to handle exceptions.

Full abstraction of the compilation scheme is preserved since the low-level is not extended with functionality that the high-level lacks. Only exceptions of existing types can be thrown and handling exceptions follows the normal course of the stack. The external code could replace an exception with a fake one, but this is equivalent to the high-level language functionality to catching an exception and throwing another one. Thus the low-level is not granted additional functionality.

3.4 Secure Compilation of Inner Classes

Inner classes are classes that are defined inside another class, as in Listing 1.4. Inner classes have access to private fields of the class they are defined within.

```
1 class AccountClass implements PI.Account {
2   AccountClass() { counter = 0; }
3   private counter : Int;
4
5   class Inner { // Inner has access to counter }
6 }
```

Listing 1.4. Example of an inner class.

Inner classes of the secure component are compiled as normal classes in the protected memory partition, in the usual fashion. To implement access from the inner class to the private fields of the surrounding class, a getter and a setter for each of its private fields are created. In the case of Listing 1.4, class `AccountClass` is extended with getters and setters for the `counter` field when compiled. Access from `Inner` to `counter` is compiled as method calls via the getter and setter.

This approach is inspired by Abadi [1], who shows that it breaks full abstraction of compilation in an early version of the JVM. In that setting, the additional low-level methods are not available at the high level, thus other low-level code other than the inner classes can call those methods, achieving something that was not possible at the high level. In our secure compilation scheme, the additional methods are available in the surrounding class. However the additional methods are not made available through entry points, thus the external code cannot invoke them. This means that the addition of inner classes to the secure compilation scheme preserves the full abstraction property.

4 Full Abstraction of the Compilation Scheme

This section presents an outline of the proof of full abstraction of the compilation scheme of Section 3. As mentioned in Section 1, a fully abstract compilation scheme preserves and reflects contextual equivalence of high- and low-level programs. This paper does not argue about the choice of contextual equivalence for modelling security properties [1,2,3,4,7,9,12].

Informally speaking, two programs C_1 and C_2 are contextually equivalent if they behave the same for all possible evaluation contexts they interact with. An evaluation context $\mathbb{C}$ can be thought of as a larger program with a hole. If the hole is filled either with C_1 or C_2, the behaviour of the whole program does not vary. Formally, contextual equivalence is defined as: $C_1 \simeq C_2 \triangleq \forall \mathbb{C}.\ \mathbb{C}[C_1]\Uparrow \iff \mathbb{C}[C_2]\Uparrow$, where $\Uparrow$ denotes divergence [20].

Denote the result of compiling a component C as $C^{\downarrow}$. Full abstraction of the compilation scheme is formally expressed as: $C_1 \simeq C_2 \iff C_1^{\downarrow} \simeq C_2^{\downarrow}$. The co-implication is split in two cases. The direction $C_1^{\downarrow} \simeq C_2^{\downarrow} \Rightarrow C_1 \simeq C_2$ states that the compiler outputs low-level programs that behave as the corresponding source programs. This is what most compilers achieve, at times even certifying the result [5,13]; we are not interested in this direction. This is thus assumed, the consequences of this assumption are made explicit (Assumption 1 below). The direction $C_1 \simeq C_2 \Rightarrow C_1^{\downarrow} \simeq C_2^{\downarrow}$ states that high-level properties are preserved through compilation to the low level. Proving this direction requires reasoning about contexts, which is notoriously difficult [4]. This is even more so in this setting, where low-level contexts are memories lacking any inductive structure. To avoid working with contexts, we equip the low-level language with a trace semantics that is equivalent to its operational semantics [18] (Proposition 1 below) and prove the contrapositive: $\mathsf{Traces}_{\mathsf{L}}(C_1^{\downarrow}) \neq \mathsf{Traces}_{\mathsf{L}}(C_2^{\downarrow}) \Rightarrow C_1 \not\simeq C_2$. This proof is based on an algorithm that creates a high-level component, a "witness" that differentiates C_1 from C_2, given that they have different low-level traces $\overline{\alpha_1}$ and $\overline{\alpha_2}$. This proof strategy is known [3,10], its complexity resides in handling features of the high-level language such as typing or dynamic memory allocation.

This proof, as well as the formalisation of Java Jr. and the assembly language, can be found in the companion report [19]. To further support the validity of this proof, the algorithm has been implemented in Scala, and it outputs Java components that adhere to the Java Jr. formalisation.[1]

Assumption 1 (Compiler preserves behaviour) *The compiler is assumed to output low-level programs that behave as the corresponding input program. Thus a high-level expression is translated into a list of low-level instructions that preserve the behaviour. By this, we mean that the following properties hold:*

- $C_1^{\downarrow} \simeq C_2^{\downarrow} \Rightarrow C_1 \simeq C_2$.
- *There exists an equivalence relation between high-level states and low-level states, such that:*

[1] Available at http://people.cs.kuleuven.be/∼marco.patrignani/Publications.html.

- *The initial high- and low-level states are equivalent.*
- *Given two equivalent states and two corresponding internal transitions, the states these transitions lead to are equivalent. Moreover, given two equivalent states and two equivalent actions, the states these transitions lead to are equivalent.*

Proposition 1 (Trace semantics is equivalent to operational semantics [18]). *For any two low-level components $C_1^{\downarrow}$ and $C_2^{\downarrow}$ obtained from compiling Java Jr. components C_1 and C_2 with the secure compilation scheme, we have that:* $\mathsf{Traces}_{\mathsf{L}}(C_1^{\downarrow}) = \mathsf{Traces}_{\mathsf{L}}(C_2^{\downarrow}) \iff C_1^{\downarrow} \simeq C_2^{\downarrow}$.

Theorem 1 (Differentiation of components). *Any two high-level components C_1 and C_2 that exhibit two different low-level trace semantics are not contextually equivalent. Formally:* $\mathsf{Traces}_{\mathsf{L}}(C_1^{\downarrow}) \neq \mathsf{Traces}_{\mathsf{L}}(C_2^{\downarrow}) \Rightarrow C_1 \not\simeq C_2$.

Theorem 2 (Full abstraction of the compilation scheme). *For any two high-level components C_1 and C_2, we have (assuming there is no overflow of the secure stack or of the secure heap):* $C_1 \simeq C_2 \iff C_1^{\downarrow} \simeq C_2^{\downarrow}$.

5 Benchmarks

This section presents benchmarking of the overhead of the secure compiler, which is proportional to the amount of boundaries crossing.

As a target low-level architecture we chose Fides [3,22]. The Fides architecture implements precisely the protection mechanism described in Section 2.1 in a very reduced TCB: ∼7000 lines of code. Fides consists of a hypervisor that runs two virtual machines: one handles the secure memory partition and one handles the other [22]. One consequence is that switching between the two virtual machines of Fides (performing calls and callbacks) is a costly operation.

For the benchmarks, we implemented a secure runtime in C. The secure runtime adds the checks presented in Section 3 to calls, callbacks (both with different number of parameters, ranging from one to eight), returns and returnbacks. These operations are executed on stub objects. A stub objects is a data structure that models the low-level representation of objects; it has an `integer` field that indicates the class of the object followed by the fields of the object. The secure runtime also contains the data structure $\mathcal{O}$ and functions that mask object references through it. Each operation was tested 1000 times on a MacBook Pro with a 2.3 GHz Intel Core i5 processor and 4GB 1333MHz DDR3 RAM. The overhead introduced for each operation ranged from 0.09% to 7.89%, averaging a 3.25% overhead. Details of the measurements can be found in the companion report [19].

6 Limitations

This section presents limitations of the compilation scheme of Section 3 and discusses garbage collection when part of the program is compiled securely.

Like many model languages [2,9], Java Jr. lacks features that real-world programming languages have, such as multithreading, foreign-function interfaces and garbage collection. A thorough investigation of the changes needed in order to support secure compilation of languages with those features is left for future work. Let us now informally discuss how garbage collection can be achieved in concert with a secure compiler.

Garbage collection is a runtime addition that handles whole programs. Firstly, assume that the external code is well-behaved and it does not disrupt the garbage collector, such as by introducing fake pointers. To perform garbage collection when part of the whole program is securely compiled, a part of the garbage collector must be trusted and allocated in the protected memory partition so that it can access $\mathcal{O}$. In this way the garbage collector can traverse the whole object graph and identify the location of a reference that is an index of $\mathcal{O}$.

Assume now that external code can disrupt the functionality of the garbage collector. The classical notion of garbage collection becomes void. In this setting the securely compiled component can be extended with a secure memory manager in charge of the secure memory partition. Here, an arguable safe methodology is to not deallocate a reference that is passed from the secure component to external code, a fact that creates problems when the allocated object is large or when many objects are passed out. In order to provide a solution to part of the problem, the compiler can introduce leasing [8]; this gives objects that are leaked a lifetime duration which, upon expiration, causes object deallocation. Alternatively, the caretaker pattern can be introduced. Instead of leaking an object reference o, the reference is wrapped in a proxy p (the caretaker) and the reference to p is leaked. In addition to method proxies for methods of o, p has a method to set the reference to o to `null`, allowing the secure memory manager to free o's memory. The problem that arises now is a breach in full abstraction: the caretaker pattern must be lifted to the high level to preserve full abstraction.

7 Related Work

This paper extends the work of Agten *et al.* [3], where the same result is achieved, but for a simpler, object-based, high-level language. This work adopts an object-oriented language with dynamic object allocation, cross-package inheritance, exceptions and inner classes, which makes the result significantly harder to achieve.

Secure compilation through full abstraction was pioneered by Abadi [1], where, alongside a result in the π-calculus setting, Java bytecode compilation in the early JVM is shown to expose methods used to access private fields by private inner classes. Kennedy [12] listed six full abstraction failures in the compilation to .NET, half of which have been fixed in modern $C^{\#}$ implementations.

Address space layout randomisation has been adopted by Abadi and Plotkin [2] and subsequently by Jagadeesan *et al.* [9] to guarantee probabilistic full abstraction of a compilation scheme. In both works the low-level language is more high-level than ours and the protection mechanism is different. Compilation does not necessarily need to target machine code, as Fournet *et al.* [7] show

by providing a fully abstract compilation scheme from an ML dialect named F* to JavaScript that relies on type-based invariants. Similarly, Ahmed and Blume [4] prove full abstraction of a continuation-passing style translation from simply-typed λ-calculus to System F. In both works, the low-level language is typed and more high-level than ours. The checks introduced by our compilation scheme seem simpler than the checks of Fournet *et al.*

A large amount of work on secure compilation applies to unsafe languages such as C, as surveyed by Younan *et al.* [24]. That research is devoted to strengthening the run-time of C and not on fully abstract compilation.

A different area of research provides security architectures with fine-grained low-level protection mechanisms. Different security architectures with access control mechanisms comparable to ours have been developed in recent years: TrustVisor [14], Flicker [15], Nizza [21], SPMs [17,22][2] and the Intel SGX [16]. The existence of industrial prototypes underlines the feasibility of this approach to bringing efficient, secure, low-level memory access control in commodity hardware. No results comparable to ours were proven for these systems.

8 Conclusion and Future Work

This paper presented a fully abstract compilation scheme for a strongly-typed, single-threaded, component-based, object-oriented programming language with dynamic memory allocation, exceptions, cross-package inheritance and inner classes to untyped machine code enhanced with a low-level protection mechanism. Full abstraction of the compilation scheme is proven correct, guaranteeing preservation and reflection of contextual equivalence between high-level components and their compiled counterparts. From the security perspective this ensures that low-level attackers are restricted to the same capabilities high-level attackers have. To the best of our knowledge, this is the first result of its kind for such an expressive high-level language and such a powerful low-level one.

Future work includes extending the results to a language with more real-world programming language features such as concurrency and distribution.

References

1. Martín Abadi. Protection in programming-language translations. In *Secure Internet programming*, pages 19–34. Springer-Verlag, London, UK, 1999.
2. Martín Abadi and Gordon Plotkin. On protection by layout randomization. In *CSF '10*, pages 337–351. IEEE, 2010.
3. Pieter Agten, Raoul Strackx, Bart Jacobs, and Frank Piessens. Secure compilation to modern processors. In *CSF '12*, pages 171–185. IEEE, 2012.
4. Amal Ahmed and Matthias Blume. An equivalence-preserving CPS translation via multi-language semantics. *SIGPLAN Not.*, 46(9):431–444, September 2011.
5. Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. *SIGPLAN Not.*, 42(6):54–65, June 2007.

[2] More thoroughly described at: https://distrinet.cs.kuleuven.be/software/pcbac

6. Roland Ducournau. Implementing statically typed object-oriented programming languages. *ACM Comput. Surv.*, 43(3):18:1–18:48, April 2011.
7. Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. Fully abstract compilation to JavaScript. In *POPL '13*, pages 371–384, New York, NY, USA, 2013. ACM.
8. C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. *SIGOPS Oper. Syst. Rev.*, 23(5):202–210, 1989.
9. Radha Jagadeesan, Corin Pitcher, Julian Rathke, and James Riely. Local memory via layout randomization. In *CSF '11*, pages 161–174. IEEE, 2011.
10. Alan Jeffrey and Julian Rathke. A fully abstract may testing semantics for concurrent objects. *Theor. Comput. Sci.*, 338(1-3):17–63, June 2005.
11. Alan Jeffrey and Julian Rathke. Java Jr.: fully abstract trace semantics for a core Java language. In *ESOP'05*, volume 3444 of *LNCS*, pages 423–438. Springer, 2005.
12. Andrew Kennedy. Securing the .NET programming model. *Theor. Comput. Sci.*, 364(3):311–317, November 2006.
13. Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, December 2009.
14. Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient TCB reduction and attestation. In *SP '10*, pages 143–158. IEEE, 2010.
15. Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: an execution infrastructure for TCB minimization. *SIGOPS Oper. Syst. Rev.*, 42(4):315–328, April 2008.
16. Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP '13*, pages 10:1–10:1. ACM, 2013.
17. Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software Trusted Computing Base. In *Proceedings of the 22nd USENIX conference on Security symposium*. USENIX Association, 2013.
18. Marco Patrignani and Dave Clarke. Fully abstract trace semantics for low-level isolation mechanisms. Under submission, 2013.
19. Marco Patrignani, Dave Clarke, and Frank Piessens. Secure Compilation of Object-Oriented Components to Protected Module Architectures – Extended Version. CW Reports CW646, Dept. of Computer Science, K.U.Leuven, 2013.
20. Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.
21. Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. Reducing TCB complexity for security-sensitive applications: three case studies. *SIGOPS Oper. Syst. Rev.*, 40(4):161–174, April 2006.
22. Raoul Strackx and Frank Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *CCS '12*, pages 2–13. ACM Press, October 2012.
23. Marko van Dooren, Dave Clarke, and Bart Jacobs. Subobject-oriented programming. In *Formal Methods for Objects and Components*. To appear, 2013.
24. Yves Younan, Wouter Joosen, and Frank Piessens. Runtime countermeasures for code injection attacks against C and C++ programs. *ACM Computing Surveys*, 44(3):17:1–17:28, 2012.